YARD Architecture Reference

Draft version July 11, 2015

Contents

i Editorial Notes

Shorthand marginal notes indicating the status of a given feature can be decoded as follows:
"`??`" indicates ill-defined
"`%%`" indicates partial implementation
"`$$`" indicates an optional feature

Register names are presented in script: r_7

Instruction mnemonics within text appear in uppercase bold: `BRA`

Instruction mnemonics within code examples appear in lowercase bold: `bra label`

1 Introduction

The YARD-1 is a simple 32 bit[1] RISC architecture having 16 registers and using a compact 16 bit, two operand instruction format.

Names

YARD = Yet Another RISC Design

YARD-1 = first version of the YARD architecture

Y1A = initial FPGA implementation of the YARD-1

Goals

Design a small FPGA RISC processor that I would enjoy programming in assembly language.

Implementations

A synthesizable BSD-licensed VHDL core, the Y1A, provides a rudimentary implementation of the YARD-1 architecture, and is intended for embedded FPGA designs utilizing internal block memory for code and data storage.

A portable[2] no-frills absolute cross assembler is provided with the Y1A core, along with an instruction set verification testbench.

The Y1A implemention provides single cycle execution of all instructions (exclusive of unused branch delay slots).

Limitations

As of this writing, the architecture definition is incomplete, in particular the coprocessor interface and processor control set.

In addition to the above, limitations of the present Y1A implementation include:

- interrupts are currently limited to a single level-sensitive external interrupt
- presently limited to on-chip BRAM
- full barrel shifter not implemented yet[3]

Architectural Influences

general RISC flavor : MIPS, ARM

assembly syntax : Motorola

conditionals via skips : 1802, PDP-8

add/subtract with skip-no-carry/borrow : NOVA

hardware input flags : 1802

short and I/O short addressing : DSP56K

[1] Early versions had an optional 16 bit datapath configuration, which is not presently supported.
[2] Cross assembler written in Perl.
[3] The Y1A shifter currently supports all 1-bit shift/rotates, and lsl/rol of 1 or 2 bits.

2 Overview

YARD-1 Feature Summary:

- 32 bit datapath
- compact 16 bit instruction format
- 16 registers
- two operand ALU instructions: register, register | immediate
- encoded short immediates: 5 bit signed, $2^N, 2^N - 1$
- immediate prefixes:
  - `IMM12` : 12 bit signed immediate
  - `LDI` : PC relative load of 32 bit immediate
- load/store architecture
- memory operand sizes: signed/unsigned 8/16/32 bit
- data operand addressing modes:
  - register indirect
  - register offset indirect
  - stack offset
  - synthetic PC-relative and absolute modes
- SKIP based conditionals
- PC-relative branches, absolute jumps
- one branch delay slot, with selectable null

3 Programming Model

3.1 Register File

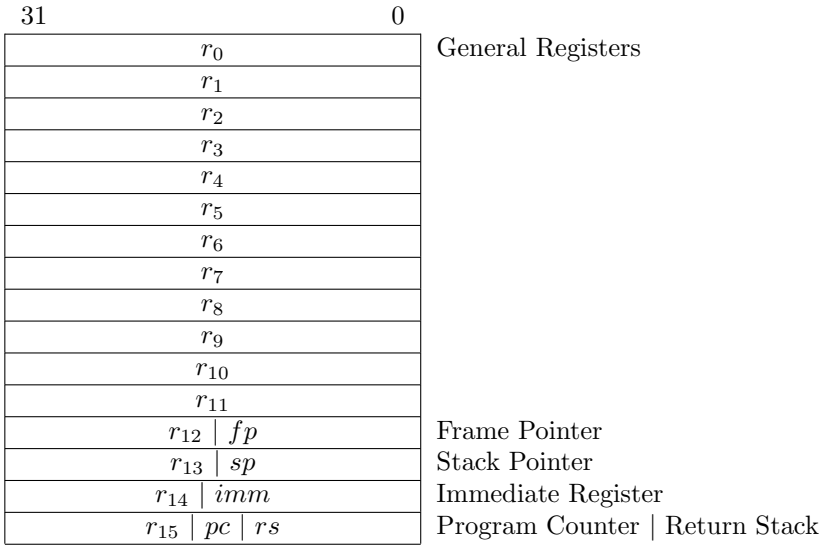

Figure 1: Register File

The first twelve registers, $r_0 \ldots r_{11}$, are completely general purpose.

Registers r_{12} and r_{13}, aka fp and sp, are available as base registers for the stack offset addressing mode, but are otherwise undedicated.

Register r_{14}, aka imm, is loadable via the `IMM12` and `LDI` opcodes to provide arbitrary immediate values (see section 5); this register also sources the offset field for offset indirect addressing. Otherwise, r_{14} may be used and referenced like any other general purpose register.

Register r_{15} is a dedicated register which provides access to either the current program counter or the hardware return stack:

- When used as the base register in address calculations (`LD, ST, LEA`), references to r_{15}, aka pc, provide the address of the current instruction.
- When accessed as a `LD/ST` data operand, references to r_{15}, aka rs, push/pop the top of the internal hardware return stack.

3.2 Special Registers

?? Special registers, for processor status and control, eventually will be accessible through the system coprocessor. See section 9.2.

3.3 Hardware Return Stack

A hardware return stack is used to store return addresses for subroutine calls; stack depth is implementation specific, with a minimum depth of 16.

The top of this return stack is accessible via `LD/ST` data operations on r_{15}, aka rs, allowing return addresses for non-leaf functions to be moved onto a software stack for unlimited call depth.

3.4 Hardware Input Flags

The processor includes 16 hardware input flags, which can be individually tested using the skip-on-flag instructions:

```
skip.fs   #N     ; SKIP Flag Set
skip.fc   #N     ; SKIP Flag Clear
```

3.5 Coprocessor Interface

??

The coprocessor interface has not been fully defined; one opcode has been reserved for coprocessor operations, allowing for 16 coprocessors, along with a skip condition slot for coprocessor conditionals.

See section 9 for more detail.

4 ALU Operations

ALU operations have the general form:

$r_a \leftarrow r_a \;\; OP \;\; op_b$ written as $OP \;\; r_a, \;\; op_b$ in assembly code

Where r_a is a register, and op_b is either a register r_b, or a short immediate constant (the short immediate encodings encompass 5 bit signed, 2^N, and $2^N - 1$ constant values).

Selectable inversion of the B operand is provided for the logical instructions, allowing for convenient bit mask generation[4] when used with the 2^N and $2^N - 1$ encoded short immediates.

mnemonic	*name*
`mov{.not}`	MOVe
`and{.not}`	AND
`or{.not}`	OR
`xor{.not}`	eXclusive OR
`add`	ADD
`sub`	SUBtract
`rsub`	Reverse SUBtract
`lsr`	Logical Shift Right
`lsl`	Logical Shift Left
`asr`	Arithmetical Shift Right
`ror`	ROtate Right
`rol`	ROtate Left
`flip`	bit FLIP
`ff1,cnt1`	(moving to coprocessor space)

Table 1: ALU Instructions

Note that in assembly code, the destination register operand r_a comes first:

```
mov     r0,r1    ; r0 = r1
add     r0,#1024 ; r0 = r0 + 1024
sub     r0,r4    ; r0 = r0 - r4
skip.gt r0,r4    ; r0 > r4
```

5 Immediates

Provision for immediates beyond the reach of the op_b short encodings is available via r_{14}, the *imm* register. *Imm* may be loaded either via any normal register operation, or by using one of the two dedicated immediate instructions:

- `imm12 #CONSTANT ; IMMediate, 12 bit`
  Loads a 12 bit sign-extended value into *imm*
- `ldi LABEL ; LoaD Immediate`
  Loads a PC Relative quad (0..4095 quad offset) into *imm*

The assembler now supports an `IMM` macro to load *imm* using the most compact encoding method (encoded short immediate `MOV`, `IMM12`, or `LDI`) when the value of the constant is known during the first pass.

```
imm     #CONSTANT       ; IMMediate, select best encoding
```

[4] 2^N single bit set `001000`; $\neg 2^N$ single bit clear `110111`; $2^N - 1$ right bit mask `000111`; $\neg(2^N - 1)$ left bit mask `111000`

6 Memory References

6.1 Operand Sizing

Memory references support three operand sizes :

- byte = 8 bits
- wyde[5] = 16 bits
- quad = 32 bits

6.2 Load and Store

Memory is byte addressable, with address alignment **REQUIRED** for 16/32 bit operands.

LD allows sign or zero extended 8/16/32 bit memory reads.

ST allows 8/16/32 bit memory writes.

6.3 Byte Order and Bit Numbering

Byte order in memory is Big-Endian, having the most significant byte of a multi-byte operand stored at the lowest memory address.

Bit numbering in registers, buses, and other multi-bit fields is Little-Bittian, wherein an N bit field has the MSB numbered N-1, and the LSB numbered zero; thus bit number N has a numerical weight of 2^N.

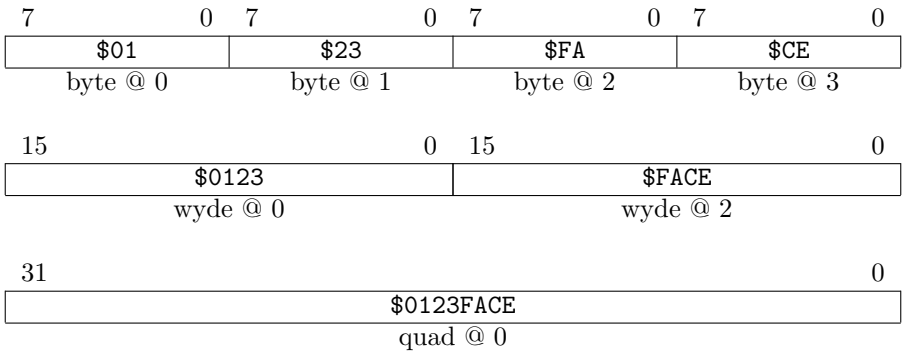

Figure 2: Operands in Memory

[5]per Knuth"MMIXware",1999,p.4: "Weight Watchers know that two nybbles make one byte, but two bytes make one wyde"

6.4 Data Addressing Modes

Fundamental Addressing Modes

- Register Indirect: `(Ra)`

```
ld   r0, (r4)
```

  The effective address is sourced by register r_a .

- Register Offset Indirect: `.imm(Ra)`

```
imm12  #$3E4
ld  r0, .imm(r4)
```

  The effective address is formed from the sum $r_a + imm$.

  A prefix instruction (`IMM12` or `LDI`), or other register operation, is needed to first load imm .

?? Once the assembler supports the automatic generation of constant prefixes, the offset will be able to be supplied directly in the memory reference:

```
ld  r0, some_offset(r4)
```

- Stack Offset: `stk_offset(fp | sp)`

```
ld.q  r0, 48(sp)
```

  This addressing mode supports **ONLY** quad loads and stores.

  The *sp* and *fp* registers **MUST** be quad aligned when using this address mode.

  The stack offset is encoded directly in the instruction as a 4 bit field, yielding an offset range of 0-15 quads (in assembly code, stk_offset is a byte offset with allowed values of 0,4,8 . . . 56,60 bytes); offsets outside this range should be assembled as a normal register offset indirect access.

Synthetic Addressing Modes

The use of register {offset} indirect addressing with either *pc* or *imm* as the base register allows PC relative and absolute address modes to be synthesized, in both short (`IMM12` prefix) and long (`LDI` prefix) variants.

As the `IMM12` prefix is signed, 'short absolute' mode can reach the top and bottom 2K bytes of memory, which works well for memory and I/O in tiny embedded systems.

?? The assembler does **NOT** currently support automatic generation of these addressing modes, but when it does the syntax will look something like this:

- PC Relative: `label(pcr)`

```
lea   r0, some_label(pcr)
```

- Absolute: `address`

```
st.b   r0, $ffff_ff00
```

7 Branches & Jumps

7.1 Jump Addressing

The jump instructions (`JMP/JSR`) require a 32 bit absolute address in a register, e.g. `jmp (r1)`.

7.2 Branch Addressing

The branch instructions support PC relative addressing in both short and long variants, with instruction (16 bit) offsets of 9, 21, and 31 bits:

instruction(s)	*instruction offset*	*branch range, instructions*
`bra | bsr`	9 bit	$+255 \ldots -256$
`imm12 + lbra | lbsr`	21 bit	$+1,048,575 \ldots -1,048,576$
`ldi + lbra | lbsr`	31 bit	$+2^{30}-1 \ldots -2^{30}$
`rbra | rbsr`	31 bit	$+2^{30}-1 \ldots -2^{30}$

Table 2: Branch Offsets

The long branch variants use an `IMM12` or `LDI` prefix to load *imm* with the upper bits of the branch offset, which is combined with 9 bits from the branch instruction field to form the 21 or 31 bit instruction offset.

7.3 Delay Slots

The programming model exposes one delay slot for the change-of-flow instructions; execution of the delay slot instruction is enabled by the `.D` suffixed instruction variants, e.g. `BRA.D, JSR.D`

Note that when the delay slot is enabled, the instruction in the delay slot should not be a branch, call, or return.

8 Conditional Instructions

8.1 SKIP

Conditional execution is supported via the `SKIP` instruction, which nullifies the execution of the following instruction if the skip condition is true.

Supported `SKIP` conditions (see SCC table in section 12.5) include register-register, register-zero, and assorted bit/byte/wyde tests; the `ADD/SUB/RSUB` instructions also provide a skip-no-carry/borrow mode.

Conditional change-of-flow is obtained by skipping a branch or jump.

```
skip.ne r0,r11      ; check for match
bra     got_it
```

8.2 WHEN

An alternate mnemonic for skip, `WHEN`, generates a skip of the opposite condition sense [6] This provides conditional execution of the following instruction when the `WHEN` condition is true.

```
.loop
  ld.b     r0,(r11)

  when.z   r0
  rts

  bsr      put_ch
  bra.d    .loop
  inc      r11
```

8.3 Multi-instruction skips

%% The `SPAM` [`Skip Propagate Against Mask`] instruction, when placed in the shadow of a skip, propagates the skip test result to the following eight instructions using either AND or XOR-NOT masking modes, whereby any given instruction is skipped if the corresponding bit of the resulting skip vector is set.

`SPAM` in AND mode [$skip_n \leftarrow skip\_test$ AND $skip\_mask_n$] skips the masked instructions when the skip condition is true, and always executes the non-masked instructions, thus extending the basic skip operation to multiple instructions (which may be intertwined with non-skipped instructions).

`SPAM` in XOR-NOT mode [$skip_n \leftarrow (skip\_test$ XOR (NOT $skip\_mask_n))$ AND $trunc\_mask_n$] implements a branch free construct akin to an if-else for short instruction sequences, in which only one of the two sets of instructions is executed. *Trunc_mask* allows the XOR-NOT mode sequence length to be shortened to fewer than eight instructions.

[6] `WHEN` was inspired by the NIOS I `IFS`, which performed the same conditional inversion for the NIOS `SKPS` instruction

```
;
;  SPAM.AND, some skipped
;   - mask is scanned left to right ( D7 = first instruction, D0 = last instruction )
;   - '1' in the mask => instruction skipped as per skip condition
;   - '0' in the mask => instruction executed normally
;
;
    mov         r1,#0

    skip.a
    spam.and    #%1010_1011

    or          r1,#$1000_0000
    or          r1,#$0200_0000
    or          r1,#$0040_0000
    or          r1,#$0008_0000

    or          r1,#$0000_1000
    or          r1,#$0000_0200
    or          r1,#$0000_0040
    or          r1,#$0000_0008

    .verify     r1,#$0208_0200


;
; SPAM.XORN, length 5
;   - mask is scanned left to right ( D7 = first instruction, D0 = last instruction )
;   - XORN mask is gated by length
;   - '1' in the mask => instruction skipped as per     skip condition
;   - '0' in the mask => instruction skipped as per NOT(skip condition)
;
;
; XORN, skip always, alternating mask, spam length 5
;
    mov         r0,#0

    skip.a
    spam.xorn   #%1010_1010,#5

    or          r0,#$1000_0000
    or          r0,#$0200_0000
    or          r0,#$0040_0000
    or          r0,#$0008_0000

    or          r0,#$0000_1000
    or          r0,#$0000_0200
    or          r0,#$0000_0040
    or          r0,#$0000_0008

    .verify     r0,#$0208_0248
```

9 Coprocessor Interface

9.1 Overview

?? The coprocessor interface has not been fully defined; one opcode has been reserved for coprocessor operations, along with a skip slot for coprocessor conditionals.

Provision exists for up to 16 coprocessors, with the first eight intended for system and standardized usage (control, memory, math, etc.), and the last eight available for user extensions.

The coprocessors have read visibility into the current register file state, allowing coprocessor calls to resemble normal function calls.

system	0 : System Control
mmu	1 : Memory Management Unit
–	2 : reserved
–	3 : reserved
imath	4 : Integer Math
–	5 : reserved
fpu	6 : Floating Point Unit
–	7 : reserved
(*user*)	8 : User defined
(*user*)	9 : User defined
(*user*)	10 : User defined
(*user*)	11 : User defined
(*user*)	12 : User defined
(*user*)	13 : User defined
(*user*)	14 : User defined
(*user*)	15 : User defined

Figure 3: Coprocessors

9.2 Coprocessor 0 : Processor Control

?? Likely processor control registers include:

31 0	
status	Status Register
control	Control Register
tick_msw	Clock Ticks (MSW)
tick_lsw	Clock Ticks (LSW)
type	Processor Type
options	Processor Options (optional instructions, stack depth)
id_msw	ID (MSW)
id_lsw	ID (LSW)

Figure 4: Processor Control Registers

10 Calling Conventions

??

Preliminary, subject to change.

The following conventions are used by the experimental YARD LCC backend[7], which was derived from the existing MIPS and ARM LCC backends; hence the similarities to the calling conventions of those processors.

The frame pointer is not currently used by the compiler, but it is marked as callee-save (a la MIPS) in the event it is pressed into use.

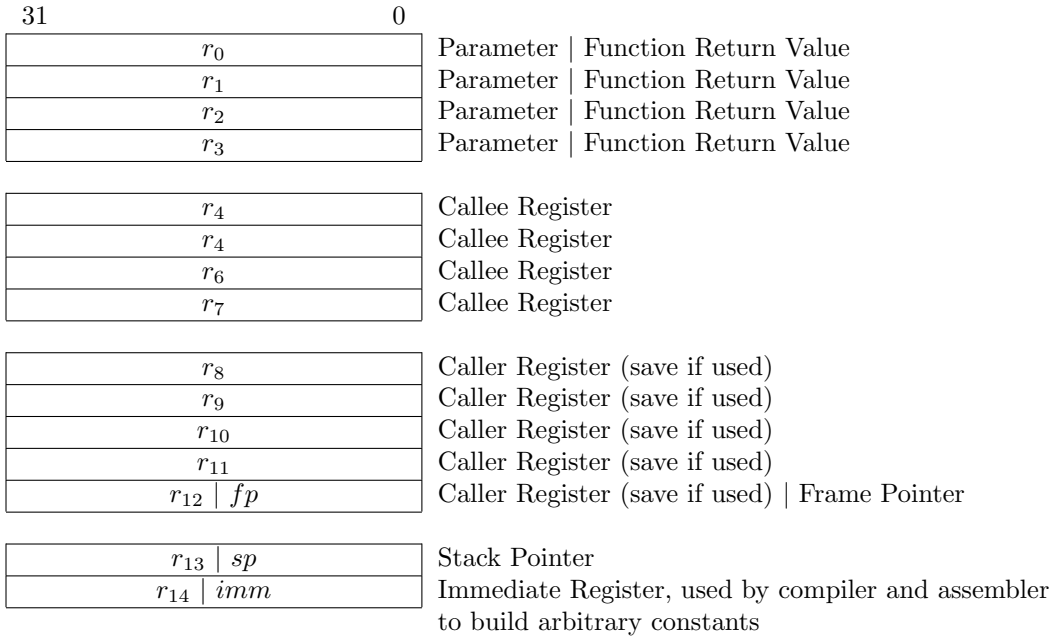

Figure 5: Calling Convention Register Usage

[7] https://code.google.com/p/lcc-homebrew/

11 Instruction Summary (Alphabetic by Mnemonic)

Opcode	Mnemonic	Name
01000ttbbbbbaaaa	add	ADD
01001ttbbbbbaaaa	add.snc	ADD, Skip No Carry
00010ttbbbbbaaaa	and	AND
00011ttbbbbbaaaa	and.not	AND NOT
0111010bbbbbaaaa	asr	Arithmetical Shift Right
1110001rrrrrrrrr	bra	BRAnch
1110000rrrrrrrrr	bra.d	BRAnch, Delayed
1110011rrrrrrrrr	bsr	Branch SubRoutine
1110010rrrrrrrrr	bsr.d	Branch SubRoutine, Delayed
0000000100000aaaa	clr	CLeaR
0101001000001aaaa	dec	DECrement
1100011100000100	di	Disable Interrupts
1100111100000100	ei	Enable Interrupts
01111111bbbbaaaa	ext.sb	EXTend, Signed Byte
01111101bbbbaaaa	ext.sw	EXTend, Signed Wyde
01111110bbbbaaaa	ext.ub	EXTend, Unsigned Byte
01111100bbbbaaaa	ext.uw	EXTend, Unsigned Wyde
0111011bbbbbaaaa	flip	FLIP
xxxxxxxxxxxxxxxx	imm	IMMediate, choose best encoding
1011iiiiiiiiiiii	imm12	IMMediate, 12 bit
0100001000001aaaa	inc	INCrement
1111101000000aaaa	jmp	JuMP
1111100000000aaaa	jmp.d	JuMP, Delayed
1111111000000aaaa	jsr	Jump SubRoutine
1111110000000aaaa	jsr.d	Jump SubRoutine, Delayed
1110101rrrrrrrrr	lbra	Long BRAnch
1110100rrrrrrrrr	lbra.d	Long BRAnch, Delayed
1110111rrrrrrrrr	lbsr	Long Branch SubRoutine
1110110rrrrrrrrr	lbsr.d	Long Branch SubRoutine, Delayed
1000mss0bbbbaaaa	ld	LoaD
1000mss1bbbbaaaa	ld.b	LoaD, Byte
1000mss0bbbbaaaa	ld.q	LoaD, Quad
1000mss1bbbbaaaa	ld.sb	LoaD, Signed Byte
1000mss1bbbbaaaa	ld.sw	LoaD, Signed Wyde
1000mss0bbbbaaaa	ld.ub	LoaD, Unsigned Byte
1000mss0bbbbaaaa	ld.uw	LoaD, Unsigned Wyde
1000mss1bbbbaaaa	ld.w	LoaD, Wyde
1010rrrrrrrrrrrr	ldi	LoaD Immediate
1001mss1bbbbaaaa	lea	Load Effective Address
0111001bbbbbaaaa	lsl	Logical Shift Left
0111000bbbbbaaaa	lsr	Logical Shift Right
00000ttbbbbbaaaa	mov	MOVe
00001ttbbbbbaaaa	mov.not	MOVe NOT
0110001000000aaaa	neg	NEGate
0000000000000000	nop	No OPeration
0011001111111aaaa	not	NOT
00100ttbbbbbaaaa	or	OR
00101ttbbbbbaaaa	or.not	OR NOT

```
Opcode                 Mnemonic   Name
-------------------------------------------------------------------
0111101bbbbbaaaa            rol   ROtate Left
0111100bbbbbaaaa            ror   ROtate Right
01100ttbbbbbaaaa           rsub   Reverse SUBtract
01101ttbbbbbaaaa       rsub.snb   Reverse SUBtract, Skip No Borrow
1111111000100000            rti   ReTurn from Interrupt
1111101000100000            rts   ReTurn from Subroutine
1111100000100000          rts.d   ReTurn from Subroutine, Delayed
1101010000000000           skip   SKIP
1101010000000000         skip.a   SKIP Always
110101010110aaaa       skip.abm   SKIP Any Byte Minus
110101010010aaaa       skip.abz   SKIP Any Byte Zero
110101010101aaaa       skip.awm   SKIP Any Wyde Minus
110101010001aaaa       skip.awz   SKIP Any Wyde Zero
1101111bbbbbaaaa        skip.bc   SKIP Bit Clear
1101011bbbbbaaaa        skip.bs   SKIP Bit Set
11010100bbbbaaaa        skip.eq   SKIP Equal
110111010111aaaa        skip.fc   SKIP Flag Clear
110101010111aaaa        skip.fs   SKIP Flag Set
11011010bbbbaaaa        skip.ge   SKIP Greater than or Equal
11011011bbbbaaaa        skip.gt   SKIP Greater Than
110111010100aaaa       skip.gtz   SKIP Greater than Zero
11011001bbbbaaaa        skip.hi   SKIP Higher
11011000bbbbaaaa        skip.hs   SKIP Higher or Same
11010011bbbbaaaa        skip.le   SKIP Less than or Equal
110101010100aaaa       skip.lez   SKIP Less than or Equal Zero
11010000bbbbaaaa        skip.lo   SKIP Lower
11010001bbbbaaaa        skip.ls   SKIP Lower or Same
11010010bbbbaaaa        skip.lt   SKIP Less Than
110101111111aaaa        skip.mi   SKIP Minus
1101110000000000         skip.n   SKIP Never
110111010110aaaa       skip.nbm   SKIP No Byte Minus
110111010010aaaa       skip.nbz   SKIP No Byte Zero
11011100bbbbaaaa        skip.ne   SKIP Not Equal
110111010101aaaa       skip.nwm   SKIP No Wyde Minus
110111010001aaaa       skip.nwz   SKIP No Wyde Zero
110111010000aaaa        skip.nz   SKIP NonZero
110111111111aaaa        skip.pl   SKIP Plus
110101010000aaaa         skip.z   SKIP Zero
11110nnnmmmmmmmm      spam.and   Skip Propagate Against Mask, AND mode
11110nnnmmmmmmmm     spam.xorn   Skip Propagate Against Mask, XOR-Not mode
1001mss0bbbbaaaa            st   STore
1001mss0bbbbaaaa          st.b   STore, Byte
1001mss0bbbbaaaa          st.q   STore, Quad
1001mss0bbbbaaaa          st.w   STore, Wyde
01010ttbbbbbaaaa           sub   SUBtract
01011ttbbbbbaaaa       sub.snb   SUBtract, Skip No Borrow
```

Opcode	Mnemonic	Name
1101110000000000	when	WHEN
1101110000000000	when.a	WHEN Always
110111010110aaaa	when.abm	WHEN Any Byte Minus
110111010010aaaa	when.abz	WHEN Any Byte Zero
110111010101aaaa	when.awm	WHEN Any Wyde Minus
110111010001aaaa	when.awz	WHEN Any Wyde Zero
1101011bbbbbaaaa	when.bc	WHEN Bit Clear
1101111bbbbbaaaa	when.bs	WHEN Bit Set
11011100bbbbaaaa	when.eq	WHEN Equal
110101010111aaaa	when.fc	WHEN Flag Clear
110111010111aaaa	when.fs	WHEN Flag Set
11010010bbbbaaaa	when.ge	WHEN Greater than or Equal
11010011bbbbaaaa	when.gt	WHEN Greater Than
110101010100aaaa	when.gtz	WHEN Greater than Zero
11010001bbbbaaaa	when.hi	WHEN Higher
11010000bbbbaaaa	when.hs	WHEN Higher or Same
11011011bbbbaaaa	when.le	WHEN Less than or Equal
110111010100aaaa	when.lez	WHEN Less than or Equal Zero
11011000bbbbaaaa	when.lo	WHEN Lower
11011001bbbbaaaa	when.ls	WHEN Lower or Same
11011010bbbbaaaa	when.lt	WHEN Less Than
110111111111aaaa	when.mi	WHEN Minus
1101010000000000	when.n	WHEN Never
110101010110aaaa	when.nbm	WHEN No Byte Minus
110101010010aaaa	when.nbz	WHEN No Byte Zero
11010100bbbbaaaa	when.ne	WHEN Not Equal
110101010101aaaa	when.nwm	WHEN No Wyde Minus
110101010001aaaa	when.nwz	WHEN No Wyde Zero
110101010000aaaa	when.nz	WHEN NonZero
110101111111aaaa	when.pl	WHEN Plus
110111010000aaaa	when.z	WHEN Zero
00110ttbbbbbaaaa	xor	eXclusive OR
00111ttbbbbbaaaa	xor.not	eXclusive OR NOT

	DIRECTIVE	.verify	.VERIFY
	DIRECTIVE	align	ALIGN
	DIRECTIVE	end	END
	DIRECTIVE	equ	EQUate
	DIRECTIVE	org	ORiGin
	DIRECTIVE	dc.b	Define Constant, Byte
	DIRECTIVE	dc.q	Define Constant, Quad
	DIRECTIVE	dc.s	Define Constant, String
	DIRECTIVE	dc.w	Define Constant, Wyde
	DIRECTIVE	dc.z	Define Constant, Zero terminated string

12 Instruction Encoding

12.1 Logical and Move Group

```
15:12  11     10:9     8:4   3:0      Operation
-------------------------------------------------------------------------
0000   NOTB   OPB_CTL  OPB    RA       MOV        RA  =          {NOT} OPB
0001   NOTB   OPB_CTL  OPB    RA       AND        RA  = RA  AND {NOT} OPB
0010   NOTB   OPB_CTL  OPB    RA       OR         RA  = RA  OR  {NOT} OPB
0011   NOTB   OPB_CTL  OPB    RA       XOR        RA  = RA  XOR {NOT} OPB
```

Abbreviations:

```
RA      : 4 bit register field (source 1 & destination)

OPB     : 5 bit immediate/4 bit register field (source 2)

OPB_CTL : select OPB type
           00: register
           01: 5 bit sign extended immediate
           10: 2^N immediate
           11: 2^N - 1 immediate

NOTB    : if set, use NOT OPB
```

12.2 Arithmetic Group

```
15:12  11     10:9     8:4   3:0      Operation
-------------------------------------------------------------------------
0100   CSKIP  OPB_CTL  OPB    RA       ADD        RA  = RA  + OPB
0101   CSKIP  OPB_CTL  OPB    RA       SUB        RA  = RA  - OPB
0110   CSKIP  OPB_CTL  OPB    RA       RSUB       RA  = OPB - RA

15:12  11:10   9       8:4   3:0      Operation
-------------------------------------------------------------------------
0111    00     0        N     RA       LSR        RA  = RA LSR  N
0111    00     1        N     RA       LSL        RA  = RA LSL  N
0111    01     0        N     RA       ASR        RA  = RA ASR  N
0111    01     1        N     RA       FLIP(*)    RA  = RA FLIP N
0111    10     0        N     RA       ROR        RA  = RA ROR  N
0111    10     1        N     RA       ROL        RA  = RA ROL  N

15:12  11:10   9:8     7:4   3:0      Operation
-------------------------------------------------------------------------
0111    11     00       RB    RA       EXT.UW     RA = zero extend RB[15:0]
0111    11     01       RB    RA       EXT.SW     RA = sign extend RB[15:0]
0111    11     10       RB    RA       EXT.UB     RA = zero extend RB[ 7:0]
0111    11     11       RB    RA       EXT.SB     RA = sign extend RB[ 7:0]
```

Note : Bit Counting instructions (CNT1,FF1) displaced by EXT are moving to coprocessor space

Abbreviations:

```
RA      : 4 bit register field (source 1 & destination)
RB      : 4 bit register field (source 2)
OPB     : 5 bit immediate/4 bit register field (source 2)

OPB_CTL : select OPB type
           00: register
           01: 5 bit sign extended immediate
           10: 2^N immediate
           11: 2^N - 1 immediate

CSKIP   : if set, skip the next instruction if NO carry/borrow occurred
          if cleared, normal execution of following instruction

N       : 5 bit field
            - bit count for shift and rotate instructions

            - bit swap enable for FLIP
               N(4) : swap even/odd wydes
               N(3) : swap even/odd bytes
               N(2) : swap even/odd nybbles
               N(1) : swap even/odd bit pairs
               N(0) : swap even/odd bits
             e.g.
               11000 : byte reverse register ( swap wydes & bytes )
               11111 : bit reverse register  ( swap everything )
               00111 : bit reverse all bytes in register
```

[7] universal bit reverse per H. Warren, "Hacker's Delight", 2003, page 102, FLIP instruction credited to Guy Steele

12.3 Memory Group

```
15:12  11   10:9   8    7:4  3:0      Operation
-------------------------------------------------------------------------------
1000  MODE  SIZE  SIGN   RB   RA      LD        RA  = {extend} [ RB {+ IMM } ]
1001  MODE  SIZE    0    RB   RA      ST        [ RB {+ IMM } ] = {trunc}  RA

1001  MODE  SIZE    1    RB   RA      LEA       RA  = RB { + IMM}


15:12                      11:0      Operation
-------------------------------------------------------------------------------
1010                       EA12      LDI       LoaD Immediate  IMM = [EA12*4 + PC & $FFFF_FFFC]


15:12                      11:0      Operation
-------------------------------------------------------------------------------
1011                       I12       IMM12     IMM = sign extend I12
```

Abbreviations:

```
RA     : 4 bit register field (destination)
RB     : 4 bit register field (address)

SIGN   : when set, sign extend memory operand on load

SIZE   : load/store operand size
           00  quad (32 bit)  stack offset addressing
           01  quad (32 bit)
           10  wyde (16 bit)
           11  byte (8 bit)

MODE   : 0 = no offset,  1 = use IMM register   for normal addressing modes
         0 = fp          1 = sp                 for stack offset addressing mode

I12    : signed 12 bit immediate for IMM12

EA12   : unsigned 12 bit quad offset for LDI
```

12.4 Control Group

15:12	11	10:9	8	7:4	3:0	Operation	
1100	?	?	?	?	?	CP	coprocessor

15:12	11:0	Operation	
1101	SCC	SKIP.CC	if (cond) skip next instruction

15:12	11	10	9	8:0	Operation	
1110	LBRA	PSHPC	DNULL	I9	{L}BRA/{L}BSR	{ push PC };
						PC = PC + ({IMM << 10} + I.9) * 2

15:12	11	10:8	7:0	Operation	
1111	0	TRUNC	MASK	SPAM.AND|XORN	skip propagate

15:12	11	10	9	8	7:4	3:0	Operation	
1111	1	PSHPC	DNULL	0	0000	RA	JMP/JSR	{ push PC }; PC = RA
1111	1	PSHPC	DNULL	0	0001	RA	RBRA/RBSR	{ push PC }; PC = PC + RA
1111	1	0	DNULL	0	0010	0000	RTS	pop PC;
1111	1	1	DNULL	0	0010	0000	RTI	restore PC1; restore PC2 & SR

15:12	11	10	9	8	7:0	Operation	
1111	1	1	DNULL	1	TR	TRAP	push PC,SR; PC = vector(TR)

Abbreviations:

```
RA     : 4 bit register field (source 1)

DNULL  : 1 = nullify delay slot instruction
         0 = execute delay slot instruction

PSHPC  : push PC

LBRA   : enables long branch: augments IMM9 offset with IMM << 10

I9     : signed nine bit instruction offset for BRA
          ( for LBRA, I9 is the low 9 bits of a 21 | 31 bit signed offset )

SCC    : skip condition field, see table in next section

TRAP   : 8 bit trap vector

MASK   : skip propagate mask
            1 = skip enable
           D7 = next instruction .. D0 = eighth instruction

TRUNC  : skip propagate truncate field
           7    : AND mode
           6..0 : XOR NOT mode, truncate skip mask by N instructions
```

12.5 Skip Condition Field

```
SCC    : skip condition

         11  10:8  7:4  3:0
        ------------------------------------------------------------------------
          0  000    RB   RA   skip.lo    lower                    unsigned, RA <  RB
          1  000    RB   RA   skip.hs    higher or same           unsigned, RA >= RB

          0  001    RB   RA   skip.ls    lower or same            unsigned, RA <= RB
          1  001    RB   RA   skip.hi    higher                   unsigned, RA >  RB

          0  010    RB   RA   skip.lt    less than                signed,   RA <  RB
          1  010    RB   RA   skip.ge    greater than or equal    signed,   RA >= RB

          0  011    RB   RA   skip.le    less than or equal       signed,   RA <= RB
          1  011    RB   RA   skip.gt    greater than             signed,   RA >  RB

          0  100    RB   RA   skip.eq    equal                    RA == RB
          1  100    RB   RA   skip.ne    not equal                RA != RB

          0  101  0000   RA   skip.z     zero                     RA == 0
          1  101  0000   RA   skip.nz    non-zero                 RA != 0

          0  101  0001   RA   skip.awz   any wyde zero
          1  101  0001   RA   skip.nwz   no wyde zero

          0  101  0010   RA   skip.abz   any byte zero
          1  101  0010   RA   skip.nbz   no byte zero

          0  101  0011   RA              reserved
          1  101  0011   RA              reserved

          0  101  0100   RA   skip.lez   less than or equal zero  RA <= 0
          1  101  0100   RA   skip.gtz   greater than zero        RA >  0

          0  101  0101   RA   skip.awm   any wyde minus
          1  101  0101   RA   skip.nwm   no wyde minus

          0  101  0110   RA   skip.abm   any byte minus
          1  101  0110   RA   skip.nbm   no byte minus

          0  101  0111    N   skip.fs    flag N set               test input flag bits
          1  101  0111    N   skip.fc    flag N clear

          0  101  1ccc    N   skip.cpt   coprocessor N true
          1  101  1ccc    N   skip.cpf   coprocessor N false
```

(continued on next page)

```
  Aliased skips:

       skip plus/minus alias to "skip.bc/bs rn, #31" :

         11:0
       --------------------------------------------------------------------------
         11111111aaaa         skip.pl    plus                    RA >= 0
         01111111aaaa         skip.mi    minus                   RA <  0

       skip always/never alias to "skip.eq/ne r0,r0" :

         11:0
       --------------------------------------------------------------------------
         010000000000         skip.a     always
         110000000000         skip.n     never


  Skip on bit:

         11  10:9  8:4  3:0
       --------------------------------------------------------------------------
          0  11     B    RA   skip.bs    bit RA.B set
          1  11     B    RA   skip.bc    bit RA.B clear
```

Abbreviations:

```
RA     : 4 bit register field (source 1)
RB     : 4 bit register field (source 2)

N      : 4 bit input flag or coprocessor number

B      : bit number (for "skip on bit")

ccc    : coprocessor skip type
```

A YARDBUG Listing

As an example of YARD assembly code, below is the listing for YARDBUG.

Source code is located in:

`%YARD_HOME%/programs/yardbug/yardbug.s`

One of my goals for code size was to be able to fit a minimal debug monitor into 256 bytes of memory.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; YARDBUG serial debugger 0.2
;
; (C) COPYRIGHT 2001-2013,2015  Brian Davis
; All rights reserved.
;
; Redistribution and use in source and binary forms, with or without
; modification, are permitted provided that the following conditions are met:
;
;    * Redistributions of source code must retain the above copyright
;      notice, this list of conditions and the following disclaimer.
;
;    * Redistributions in binary form must reproduce the above copyright
;      notice, this list of conditions and the following disclaimer in the
;      documentation and/or other materials provided with the distribution.
;
; THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
; AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
; IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
; ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
; LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
; CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
; SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
; INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
; CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
; ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
; POSSIBILITY OF SUCH DAMAGE.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  RS232 data format: 19200,N,8,1
;  ( RS232 format is set in evb.vhd, not controlled by S/W yet )
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;commands:
;
;   D  ADDR COUNT         Dump memory
;
;   G  ADDR               Go
;
;   M  ADDR BYTE {BYTE}   Modify byte(s)
;
;   ?                     help
;
;  e.g.:
;    D 0 100         ; dump 256 bytes starting at zero
;    G 0             ; jump to address 0
;    M 600 00        ; byte at $600 = 0
;    M 601 ff ee     ; byte at $601 = $ff, $602 = $ee
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Quirks:
;
;   - parses input on the fly, no line editing:
;
;      - no way to escape out of command in progress
;
;      - no way to backspace
;        ( uses last 8/2 hex chars. entered for quad/byte input,
;          so you can type some zeroes followed by the proper
;          value to correct an error)
;
```

```
                    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                    ;
                    ; I/O addresses
                    ;
C0000000            UART    equ $c000_0000

                    ;
                    ; Input Flag bit assignments
                    ;
0000000F            FLAG_TX_RDY     equ 15
0000000E            FLAG_RX_AVAIL   equ 14


                    ;
                    ; register usage:
                    ;
                    ;   r0 = local, parameter, return value
                    ;   r1 = local, 2nd return value
                    ;   r2 = unused
                    ;   r3 = local
                    ;   r4 = local
                    ;   r5 = unused
                    ;   r6 = unused
                    ;   r7 = unused ( old command search local )
                    ;   r8 = unused
                    ;   r9 = global, UART port address
                    ;   r10= local
                    ;   r11= local
                    ;   r12= local (mem address)
                    ;   r13= local (mem loop count)
                    ;   r14= IMM, locals, address parameter
                    ;

                    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                    ;
                    ;
                    ; program start
                    ;
                    ;   $0 for embedded boot "ROM"
                    ;
                    ;   after code squish for ROM space, debugger must live in the first
                    ;   256 bytes of memory for the compact command table dispatch to work
                    ;
00000000                org     $0

00000000            start:
                    ; put NOP back in to diagnose intermittent HW reset problem
00000000  0000          nop

                    ; UART addresses
00000002  0FE9          mov     r9, #UART


                    ; print the help message
00000004  E62C          bsr     cmd_help

                    ;
                    ; main loop
                    ;
00000006            parse_loop:
00000006  E65C          bsr     send_crlf

00000008  0460          mov     r0, #'@     ; @ or ? for prompt fits into short immediate encoding
0000000A  E662          bsr     send_char

                    ; get a character & echo it
0000000C  E665          bsr     get_char_echo

                    ;
                    ; convert any alpha char. a-z to upper case A-Z
                    ;  ( this code also clobbers some punctuation chars )
                    ;
0000000E  DE60          when.bs r0, #6          ; D6 is set for all ASCII letters
00000010  1C50          and     r0, #$FFFF_FFDF ; clear D5 : $60-7f -> $40-5f


                    ;
                    ; future - add backspace check here, bra back to get another command character
                    ;
```

```
                        ;
                        ; check character in r0 against the command list
                        ;   byte-wide pointer version of table search,
                        ;   target routine must be in low 256 bytes of memory
                        ;

00000012  B0FB              imm12   #CMD_LNK     ; r13 = address of command entry point table
00000014  00ED              mov     r13, r14

00000016  B0F6              imm12   #CMD_TAB     ; r14 = address of command character lookup table

00000018                .match_loop:
00000018  86EB              ld.ub   r11, (r14)   ; r11 = next table command byte

0000001A  421E              inc     r14          ; bump r14 to next entry

0000001C  DD0B              when.z  r11          ; bail out if at end of table
0000001E  E3F4              bra     parse_loop

00000020  DCB0              when.eq r0,r11       ; check for match
00000022  E203              bra     .got_it

00000024  421D              add     r13,#1       ; increment byte address pointer
00000026  E3F9              bra     .match_loop

00000028                .got_it:
00000028  86DB              ld.ub    r11, (r13)      ; r11 = command subroutine address

0000002A  E656              bsr     get_char_echo   ; get & echo one character after command letter

0000002C  FE0B              jsr     (r11)           ; call selected command

0000002E  E3EC              bra parse_loop


                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ;
                        ; dump memory
                        ;
00000030                cmd_dump:

                        ; get the start address into R12
00000030  E623              bsr     ghex
00000032  001C              mov     r12, r1

                        ; if user entry was terminated with a space, read byte count; else dump 16 bytes
00000034  0441              mov     r1,#16       ; load default count

00000036  DD00              when.z  r0          ; check return code of last ghex call ( r0=0 for a space )
00000038  E61F              bsr     ghex         ; get user byte count

0000003A  5211              dec     r1           ; decrement byte count ( loop counter counts N-1..0 )

0000003C  001D              mov     r13, r1              ; r13 = byte count
0000003E  16DD              and     r13, #$0000_1fff     ; limit loop iterations to 8K

00000040                .next_line:
00000040  00C0              mov     r0, r12      ; print address
00000042  E628              bsr     phex32

00000044  044B              mov     r11,#16      ; load line byte counter

00000046                .dump_loop:
00000046  E643              bsr     space

00000048  86C0              ld.ub   r0, (r12)    ; read next memory byte
0000004A  E622              bsr     phex8

0000004C  421C              inc     r12          ; increment memory pointer

0000004E  5A1D              sub.snb r13,#1       ; decrement byte count, check for borrow
00000050  E237              bra     send_crlf    ; bail out if negative (tail call)

00000052  521B              dec     r11          ; decrement line byte counter

00000054  D50B              when.nz r11
00000056  E3F8              bra     .dump_loop

00000058  E633              bsr     send_crlf    ; end of line

0000005A  E3F3              bra     .next_line
```

```
                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ;
                        ; help
                        ;
0000005C                cmd_help:

                        ;
                        ; note, CRLF string and list of commands immediately follow
                        ; the banner string, so two calls to pstr will print all three
                        ;
0000005C  B0E8              imm12   #STR_BANNER
0000005E  E631              bsr     pstr        ; print initial banner
00000060  E230              bra     pstr        ; print CRLF & commands (tail call)


                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ;
                        ; modify memory
                        ;
00000062                cmd_modify:

                        ; get the address into R4
00000062  E60A              bsr     ghex
00000064  0014              mov     r4, r1

00000066                .next_byte:

                        ; get the data byte
00000066  E608              bsr     ghex

                        ; write it
00000068  9641              st.b    r1, (r4)
0000006A  4214              inc     r4

                        ; check returned ghex terminator value to see if we're done
0000006C  DD00              when.z  r0
0000006E  E3FC              bra     .next_byte   ; zero -> space, so go get another byte

00000070  FA20              rts


                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ;
                        ; go
                        ;
00000072                cmd_go:
00000072  E602              bsr     ghex    ; r1 = target address
00000074  FA01              jmp     (r1)    ; jump to user program


                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ;
                        ; utility routines
                        ;
                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


                        ;
                        ; ghex:
                        ;   read hex value from user,
                        ;
                        ;   reads as many chars as are entered, shifting result up 4 bits for each,
                        ;   until it sees a space or control character
                        ;
                        ;   returns:
                        ;     r0 = terminating char. - $20  (e.g., r0 = 0 if terminator was a space )
                        ;     r1 = value of hex entry
                        ;
00000076                ghex:

                        ; initialize working value in r1
00000076  0201              mov r1,#0

00000078                .gloop:
00000078  E62F              bsr     get_char_echo

                        ; bail out on a space or other ctl. char
0000007A  5450              sub     r0, #$20
0000007C  DD40              when.lez  r0
```

```
0000007E  FA20          rts

                        ;
                        ; convert ASCII code  -> hex ( with liberties taken for illegal chars )
                        ;
                        ; before & after subtract of $20:
                        ;   $30:$39 ( '0':'9' )  now is $10:19
                        ;   $41:$46 ( 'A':'F' )  now is $21:26
                        ;   $61:$66 ( 'a':'f' )  now is $41:46
                        ;
00000080  D640          skip.bs r0, #4  ; if D4 is set, assume it's a numeral 0-9
00000082  4290          add     r0, #9  ; else convert 'a|A'-'f|F' -> $a-$f in LS nybble

00000084  12F0          and     r0, #$0f ; keep only the 4 LSB's

                        ; shift last value up 4 bits
00000086  E617          bsr     rol_r1_four
00000088  1301          and     r1, #$ffff_fff0    ; clear out the 4 LSB's

                        ; or in new nybble
0000008A  2001          or      r1, r0

0000008C  E3F6          bra     .gloop

                        ;
                        ; print 8/16/32 bit hex value in r0
                        ;
0000008E                phex8:
0000008E  B008          imm12   #8          ; # bits
00000090  E202          bra     phex

                        ;
                        ; phex16 not currently needed, commented out to save ROM space
                        ;
                        ;phex16:
                        ;    imm12   #16
                        ;    bra     phex

00000092                phex32:
00000092  B020          imm12   #32         ; # bits

00000094                phex:
00000094  0001          mov     r1,r0       ; copy input value to R1

00000096  00E3          mov     r3,r14      ; copy length to R3 = bit counter

00000098  645E          rsub    r14,#32     ; initial rotate of r0 by 32-N bits left to move desired field into N
0000009A  E60E          bsr     rol_r1_imm

0000009C                .hloop:
0000009C  E60C          bsr     rol_r1_four ; rotate MS nybble into the LS nybble

0000009E  0010          mov     r0, r1      ; copy to r0 and mask
000000A0  12F0          and     r0, #$0f

                        ; hex to ASCII conversion
000000A2  5290          sub     r0, #9

000000A4  DD40          when.lez  r0
000000A6  5270          sub     r0, #7

000000A8  4460          add     r0, #$40

000000AA  E612          bsr     send_char   ; print character

000000AC  5243          sub     r3,#4       ; decrement bit count

000000AE  D543          when.gtz  r3
000000B0  E3F6          bra     .hloop

000000B2  FA20          rts

                        ;
                        ; rol_r1_imm
                        ;   rotate r1 imm positions left
                        ;
                        ; rol_r1_four
                        ;   alternate entry point, rotate r1 four positions left
                        ;
                        ;
000000B4                rol_r1_four
```

```
000000B4  B004          imm12   #4

000000B6              rol_r1_imm
000000B6  5A1E          sub.snb r14,#1          ; decrement shift count
000000B8  FA20          rts                     ; bail out if done

000000BA  E1FE          bra.d   rol_r1_imm      ; loop back (delayed)
000000BC  7A11          rol     r1              ; rotate one bit left

                      ;
                      ; print null terminated string
                      ;
                      ;   r14 = address of string
                      ;
                      ;   uses r0
                      ;

                      ; alternate entry point prints CRLF string
000000BE              send_crlf:
000000BE  B0F3          imm12   #STR_CRLF

000000C0              pstr:
000000C0  86E0          ld.ub  r0,(r14)     ; get next byte of string
000000C2  421E          inc    r14          ; bump pointer to next character

000000C4  DD00          when.z r0           ; bail out if zero terminator
000000C6  FA20          rts

000000C8  E603          bsr  send_char      ; send character

000000CA  E3FB          bra  pstr           ; loop


                      ;
                      ; send a character using HW UART
                      ;
                      ;   input data in r0
                      ;
                      ;   expects
                      ;     r9 = UART port address
                      ;

                      ; send a space, falls through to send_char
000000CC              space:
000000CC  0450          mov     r0, #$20

000000CE              send_char:

                      ; loop until transmitter ready flag = 1
000000CE              .tx_wait:
000000CE  D57F          skip.fs #FLAG_TX_RDY
000000D0  E3FF          bra     .tx_wait

000000D2  9290          st      r0, (r9)    ; write data to TX
000000D4  FA20          rts

                      ;
                      ; version of get_char with echo of input character
                      ;
                      ;  also handles CRLF expansion
                      ;  note funky return value (0) for a CR
                      ;
000000D6              get_char_echo:
000000D6  E605          bsr get_char

000000D8  02DA          mov     r10,#$0d

                      ; if CR echo both CR & LF
000000DA  DC0A          when.eq r10,r0
000000DC  E3F1          bra     send_crlf   ; note that send_crlf will return r0=0 to the caller in place of CR

                      ; else echo the original character
000000DE  E3F8          bra     send_char   ; (tail call)


                      ;
                      ; receive a character using HW UART
                      ;
                      ;   returns char in r0
                      ;
                      ;   expects
```

```
                        ;       r9 = UART port address
                        ;
000000E0                get_char:

                        ; loop until there's something in the buffer
000000E0                .rx_empty:
000000E0  D57E              skip.fs #FLAG_RX_AVAIL
000000E2  E3FF              bra     .rx_empty

000000E4  8290              ld      r0, (r9)    ; read data from RX
000000E6  FA20              rts


                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ;
                        ; constant tables
                        ;
                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


                        ;
                        ; constant message strings
                        ;
                        ; note, cmd_help currently requires STR_BANNER, STR_CRLF, and CMD_TAB to
                        ; immediately follow one another so calls to pstr print each in turn
                        ;
000000E8                STR_BANNER:
000000E8  59                dc.s    "YARDBUG 0.2"
000000E9  41
000000EA  52
000000EB  44
000000EC  42
000000ED  55
000000EE  47
000000EF  20
000000F0  30
000000F1  2E
000000F2  32

000000F3                STR_CRLF:
000000F3  0D                dc.b    $0d,$0a,$00
000000F4  0A
000000F5  00


                        ;
                        ; commmand table
                        ;
                        ; notes:
                        ;
                        ;   - entries in table should be uppercase letters (A-Z), or punctuation chars < ASCII code 
                        ;
                        ;   - one-byte table addresses limit calls to routines located in first 256 locations of mem
                        ;
000000F6                CMD_TAB:
000000F6  44                dc.s    "DGM?"
000000F7  47
000000F8  4D
000000F9  3F
000000FA  00                dc.b    0               ; end of table marker


                        ;       align   4           ; align not needed for byte-wide pointer table

000000FB                CMD_LNK:
000000FB  30                dc.b    cmd_dump
000000FC  72                dc.b    cmd_go
000000FD  62                dc.b    cmd_modify
000000FE  5C                dc.b    cmd_help

000000FF  00                dc.b    0               ; end of table marker



                        ; ;
                        ; ; shorter help message (save ROM space)
                        ; ;
                        ; STR_HELP:
                        ;       dc.s    "YARDBUG 0.02"
                        ;       dc.b    $0d,$0a
                        ;
```

```
                        ;       dc.s    "D a #|G a|M a b {b}|?"
                        ;
                        ; STR_CRLF:
                        ;       dc.b    $0d,$0a
                        ;       dc.b    0
                        ;

                        ; original (longer) help message
                        ;
                        ; ;
                        ; ;
                        ; ;
                        ; STR_HELP:
                        ;       dc.s "YARDBUG 0.02"
                        ;       dc.b    $0d,$0a
                        ;       dc.s    " D addr cnt"
                        ;       dc.b    $0d,$0a
                        ;       dc.s    " G addr"
                        ;       dc.b    $0d,$0a
                        ;       dc.s    " M addr byte {byte}"
                        ;       dc.b    $0d,$0a
                        ;       dc.s    " ?"
                        ; STR_CRLF:
                        ;       dc.b    $0d,$0a
                        ;       dc.b    0
                        ;


                          end


Symbols (by name):

  00000030  cmd_dump
  00000046  cmd_dump.dump_loop
  00000040  cmd_dump.next_line
  00000072  cmd_go
  0000005C  cmd_help
  000000FB  CMD_LNK
  00000062  cmd_modify
  00000066  cmd_modify.next_byte
  000000F6  CMD_TAB
  0000000E  FLAG_RX_AVAIL
  0000000F  FLAG_TX_RDY
  000000E0  get_char
  000000E0  get_char.rx_empty
  000000D6  get_char_echo
  00000076  ghex
  00000078  ghex.gloop
  00000006  parse_loop
  00000028  parse_loop.got_it
  00000018  parse_loop.match_loop
  00000094  phex
  0000009C  phex.hloop
  00000092  phex32
  0000008E  phex8
  000000C0  pstr
  000000B4  rol_r1_four
  000000B6  rol_r1_imm
  000000CE  send_char
  000000CE  send_char.tx_wait
  000000BE  send_crlf
  000000CC  space
  00000000  start
  000000E8  STR_BANNER
  000000F3  STR_CRLF
  C0000000  UART


Symbols (by value):

  00000000  start
  00000006  parse_loop
  0000000E  FLAG_RX_AVAIL
  0000000F  FLAG_TX_RDY
  00000018  parse_loop.match_loop
  00000028  parse_loop.got_it
  00000030  cmd_dump
  00000040  cmd_dump.next_line
  00000046  cmd_dump.dump_loop
  0000005C  cmd_help
```

```
  00000062  cmd_modify
  00000066  cmd_modify.next_byte
  00000072  cmd_go
  00000076  ghex
  00000078  ghex.gloop
  0000008E  phex8
  00000092  phex32
  00000094  phex
  0000009C  phex.hloop
  000000B4  rol_r1_four
  000000B6  rol_r1_imm
  000000BE  send_crlf
  000000C0  pstr
  000000CC  space
  000000CE  send_char
  000000CE  send_char.tx_wait
  000000D6  get_char_echo
  000000E0  get_char
  000000E0  get_char.rx_empty
  000000E8  STR_BANNER
  000000F3  STR_CRLF
  000000F6  CMD_TAB
  000000FB  CMD_LNK
  C0000000  UART

Total Warnings = 0
Total Errors   = 0
```